Checking Contact Tracing App Implementations

Checking Contact Tracing App Implementations

Robert Flood[1], Sheung Chi Chan[2], Wei Chen[1] and David Aspinall[1,3]
[1]*LFCS, University of Edinburgh, Edinburgh, U.K.*
[2]*Department of Computer Science, Heriott-Watt University, Edinburgh, U.K.*
[3]*The Alan Turing Institute, London, U.K.*

Keywords: Static Analysis, Covid-19, Contact Tracing, Android.

Abstract: In the wake of the COVID-19 pandemic, contact tracing apps have been developed based on digital contact tracing frameworks. These allow developers to build privacy-conscious apps that detect whether an infected individual is in close-proximity with others. Given the urgency of the problem, these apps have been developed at an accelerated rate with a brief testing period. Such quick development may have led to mistakes in the apps' implementations, resulting in problems with their functionality, privacy and security. To mitigate these concerns, we develop and apply a methodology for evaluating the functionality, privacy and security of Android apps using the Google/Apple Exposure Notification API. This is a three-pronged approach consisting of a manual analysis, general static analysis and a bespoke static analysis, using a tool we've developed, dubbed MonSTER. As a result, we have found that, although most apps met the basic standards outlined by Google/Apple, there are issues with the functionality of some of these apps that could impact user safety.

1 INTRODUCTION

As governments around the world attempt to contain the spread of the COVID-19 virus, the research area of digital contact tracing has grown rapidly with several methods being proposed to aid this cause. Digital contact tracing refers to the tracking of individuals to determine potential exposure between an infected patient and a user, using mobile technologies such as QR codes, Bluetooth and GPS. Currently, the most effective strategy to control the outbreak is widespread social-distancing and isolation of even healthy individuals. This has significantly impacted almost every aspect of daily life, with profound economic and social drawbacks. Health authorities hope digital contact tracing will allow for social-distancing measures to be eased by automating the time-consuming process of manual contact tracing, allowing more individuals to discover whether they are infected.

Such efforts may leave users vulnerable to security and privacy flaws. Due to the urgency in developing contact tracing apps, many have been built at an accelerated rate. It is unclear if measures were undertaken to minimise the risk of security vulnerabilities. As these apps need to be used by a large segment of a country's population to be effective, the integrity of many people's data and digital assets may be at risk.

So far, ensuring tracing frameworks maintain strong privacy guarantees has been the focus of research, with new privacy-preserving frameworks being designed by several parties (Google, 2020; Troncoso et al., 2020; Wan and Liu, 2020). However, little research has investigated real-world implementations of these frameworks; the apps using these frameworks can violate these privacy guarantees by sharing additional information. As digital contact tracing techniques involve the collection of sensitive medical and location data in order to function, there are severe privacy implications if this information is stolen or improperly handled. For instance, the Bahraini *BeAware* was linked to a televised game-show 'Are You At Home?', where users of the app were called and offered prizes if they were at home and shamed otherwise. This show leveraged data collected from the app such as the contestant's name and phone number (Amnesty, 2020). Such misuse could have been prevented by using a privacy-preserving framework, provided the app did not retrieve further information. Thus, it is important that a tracing app uses a privacy-preserving framework, but also it does not share data beyond what the framework allows.

As government bodies urge people to use tracing apps, they are an enticing target for malicious actors. Malware claiming to be contact tracing software

or malicious, repackaged versions of contact tracing apps exist (Anomali, 2020; ESET, 2020) and more are likely to be discovered. Developers may also introduce security problems via misuse of these frameworks. Therefore, tracing frameworks have considered potential security issues in their design. For instance, the Google/Apple Exposure Notifications (GAEN) API acts as a security boundary, allowing developers to access its functionality whilst shielding them from its internal operation. However, previous similar secure APIs have had faulty implementations that led to fundamental issues, such as the PCKS 11 API of hardware security modules leaking their private keys (Bortolozzo et al., 2010).

This paper introduces a methodology for analysing the functionality, privacy and security of COVID-19 contact tracing apps. We employ manual and static analysis alongside a bespoke, customisable static analysis tool. This bespoke tool, developed by us and dubbed MonSTER, ensures apps adhere to the requirements of a given framework, something impossible with an out-of-the-box tool, and can provide repeatable, lightweight checks throughout an app's development and updates. We target apps using the GAEN API. Our approach and analysis focus solely on the implementation of the apps themselves and assume the adopted contact tracing framework is well defined and problem-free. This helps protect against mistakes by app developers, as well as attacks and infiltration by adversaries by discovering potential security vulnerabilities. Ultimately, work like this may help assuage public concerns of using these apps and increase uptake, helping to better contain the spread of COVID-19.

The contributions of this paper are as follows:

- We design a methodology with three stages, manual analysis, off-the-shelf static analysis and bespoke static analysis, to evaluate the functionality, privacy and security of contact tracing apps.
- We develop MonSTER, a configurable, lightweight static analysis tool to verify an app's adherence to API usage requirements.
- We collect a set of 12 contact tracing apps using the GAEN API and, where necessary, rebuild the apps from their source code to obtain a non-obfuscated APK. We have released these apps as a public dataset [1].
- We obtain results demonstrating that, although the majority of apps tested functioned correctly, there were implementation problems in some apps that impacted their functionality. Namely, we found apps may incorrectly inform users contact tracing is enabled when it is in fact disabled during reasonable usage.

The structure of the rest of this paper is as follows. Section 2 provides background information, including security and privacy concerns, and an overview of the GAEN API. Section 3 describes our methodology, including our lightweight static analysis tool *MonSTER* (*Mon*oid-based *St*atic Analys*er*). Section 4 describes the process we undertake for our manual, static, and MonSTER analysis in more detail. Section 5 shows the results of this analysis. Section 6 discusses the effectiveness of our approach alongside a brief discussion on contact tracing app designs. Section 7 summarises the related work. Section 8 concludes and discusses ongoing and future steps.

2 BACKGROUND

Due to privacy concerns surrounding the collection of user location data by governments, decentralised approaches to digital contact tracing are promoted. Decentralised approaches trace contacts with minimal interaction with a central database; several have been proposed, including *TCN* (TCN, 2020), *DP3T* (Troncoso et al., 2020) and the *Google/Apple Exposure Notification* protocol or GAEN for short (Google, 2020). They are similar in design, using Bluetooth Low Energy (BLE) to measure distance between users. The effectiveness of BLE tracking has been criticised (Leith and Farrell, 2020), but tracking can be augmented with the use of QR-code registrations, and even a partially successful approach may help.

In this section, we discuss details of the GAEN Framework and potential security/privacy problems.

2.1 Google/Apple Exposure Notification

This framework is developed by Apple and Google for their mobile operating systems, each providing the same set of API calls. The major features of the GAEN Framework are divided into three stages. For each stage, we summarise the responsibilities of the app developers, the communication of the servers and the underlying APIs. We focus our research on version 1.3 of the framework.

Contact Exchange. When first setting up the app, the device generates a *Temporary Exposure Key* (TEK) using a cryptographically secure random number generator. Every 24 hours, a new TEK is generated. Two further keys are generated using this key via HKDF, a *Rolling Proximity Identifier Key* and an *Associated Encrypted Metadata Key*. These keys are in turn used to generate two BLE

[1] https://github.com/glo-fi/GAEN-Contact-Tracing-Apps

payloads: the *Rolling Proximity Identifier* (RPI) and the *Associated Encrypted Metadata* (AEM). These consist of a byte string acting as an identifier and a payload that can be later decrypted to reveal the user's TEK, both encrypted using AES-128. The phone alternates between the Bluetooth client and host modes, continuously broadcasting the RPI and AEM when in client mode and seeking such broadcasts from nearby phones when in host mode, storing any received payloads. Figure 1 shows a sequence diagram for this stage.

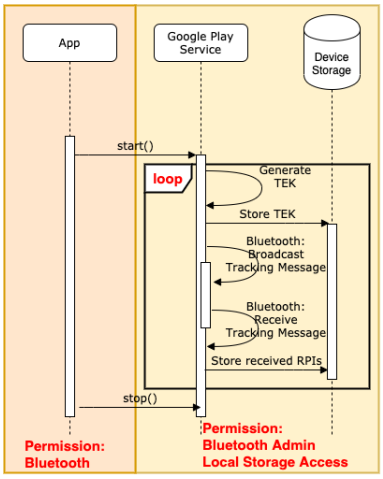

Figure 1: Contact Exchange Sequence Diagram.

Infection Report. If a user tests positive with COVID-19, they can choose to upload their TEK history, extending back 14 days, to a *Diagnosis Server*, alongside a timestamp to describe when their validity started. These are referred to as *Diagnosis Keys* and only leave the device if the user tests positive. The contact tracing app is responsible for verifying a diagnosis report from an authorised medical provider. The apps are responsible for protecting this information during the uploading process. Identifiers over 14 days old are considered no longer useful as infected user would have likely recovered in that time. Thus, identifiers are deleted from both the device and central server 14 days after being first created. Figure 2 shows a sequence diagram for this stage.

Exposure Update. To determine if a user was exposed to an infected person, users routinely download the list of newly added Diagnosis Keys from the Diagnosis Server. As RPIs are derived from TEKs, each client can then derive a series of RPIs from these Diagnosis Keys. These derived identifiers can then be matched with the list of stored

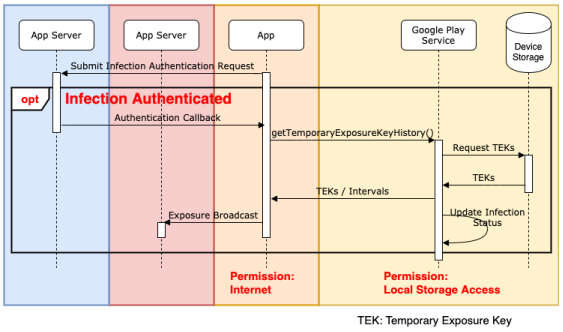

Figure 2: Infection Report Sequence Diagram.

identifiers discovered over BLE scanning. If any of the derived identifiers match a stored identifier then the user has come into contact with someone infected with COVID-19 and the app should notify them of this. The app should implement a broadcast receiver to receive this information and use an API call to retrieve further exposure summary information, notifying the users of this fact. None of these steps should reveal the identity of the infected person nor the notified users. Figure 3 shows a sequence diagram for this stage.

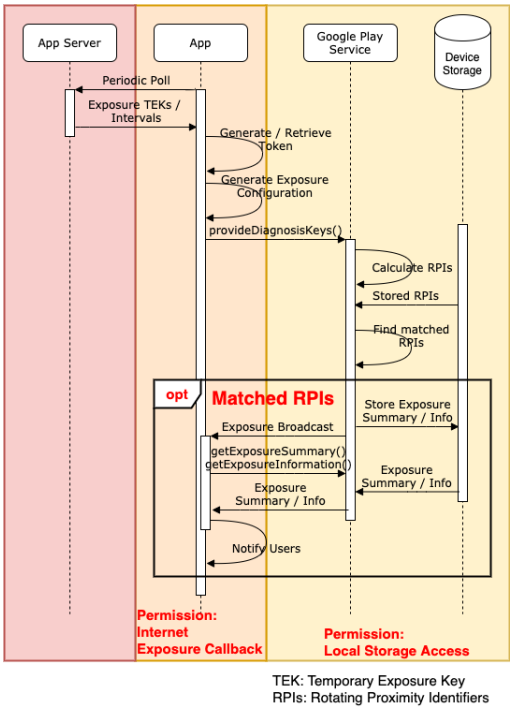

Figure 3: Exposure Update Sequence Diagram.

2.1.1 App Responsibilities

To aid the development of these apps, Google and Apple provided the GAEN API and documentation (Google, 2020) outlining its usage. They describe

the functionality the API provides and the functionality app developers must provide. The API handles the complex aspects of the protocol including the cryptographic systems, the broadcasting and collection of BLE data and the calculation of an 'exposure risk level'. The documentation states the app must:

1. Allow users to start and stop contact tracing.
2. Register a Broadcast Receiver to receive the `ACTION_EXPOSURE_STATE_UPDATED` intent
3. Poll the Diagnosis server to obtain keys.
4. Download Diagnosis Keys and provide them to the API.
5. Upload TEKs after a positive test and the user has provided permission.
6. Notify the user with medical information when they have been exposed to an infected user.

These responsibilities are part of the API lifecycle outlined in Section 2.1. To fulfill these, the GAEN Framework provides developers with an API via the `ExposureNotificationClient` class. In this paper, we focus on the API methods `start`, `stop`, `isEnabled`, `getTemporaryExposureKeyHistory` (which retrieves the past two weeks of TEKs) and `provideDiagnosisKeys` (which submits downloaded keys to the API).

2.2 Security Concerns

In developing our methodology for analysing the security, privacy and functionality of contact tracing apps, we consider two attacker models.

For the first, we assume the developers attempted to securely and faithfully adopt the GAEN framework but failed, reducing user privacy or providing exploit vectors for attackers. Although the GAEN API has been designed with privacy and security in mind, apps using this API must adhere to strict requirements in order to maintain these properties. For instance, developers accidentally misusing the GAEN API may build an app that retrieves and uploads the users' TEK history overly frequently. This results in the online database resembling a centralised protocol, such as BlueTrace, damaging user privacy. These functionality failings may lead to numerous security and privacy violations: exposure of a user's identity; their address; their employer; their infection status; the identities of their contacts; their location and the loss of security of their device, which may lead to other potential data exposures. Even if the GAEN API is correctly utilised, the app may contain vulnerabilities impacting user security, such as misconfigured webviews or exported components.

For the second, we consider the case where the app is malware purporting to perform contact tracing, perhaps as a repackaged version of a legitimate app. Such apps have already been discovered, including backdoors (Anomali, 2020) and ransomware (ESET, 2020). In this situation, the fact that the malware claims to be a contact tracing app is incidental to its true behaviour. Nevertheless, any systematic review of the security of contact tracing apps needs to consider this possibility.

The two attack models discussed above concentrate on the wrongful adoption of the GAEN API, either accidental or intentional; both result in possible security and privacy problems which leak user information. Although such security problems can occur in the GAEN framework, our approach focuses on the adoption of the framework. Thus, our attack models consider the security concerns of only the implementation of the contact tracing apps.

We discuss how these models inform our methodology for analysing contact tracing apps in Section 3.

2.3 Static Analysis Tools

There are many static analysis tools for Android software. However, these tools know little about the context in which they are applied. Many tools used for Android code analysis are primarily Java analysis tools, such as Error-Prone (Sadowski et al., 2018) and FindBugs (Ayewah et al., 2008), and are unaware of any potential problems specific to the Android platform or any particular details of the app. Many security vulnerabilities are the result of faulty implementation logic which cannot be divorced from the app's utility. As such, there are a wide-range of bugs that general static analysis tools are incapable of finding.

Contact tracing apps are one domain where general static analysis is lacking and implementation bugs may arise. Prior to March 2020, there were no tracing apps developers could base their apps on, and disparate teams are developing apps with little guidance. The challenges faced by tracing apps are unique, and it is unlikely general static analysis solutions could detect functionality issues or missing features, such as notifying the user when an exposure occurs. It would be extremely beneficial to have a static analysis tool that ensures an app is adhering to necessary standards during its development and release.

3 METHODOLOGY

In this section, we discuss our methodology, together with the static analysis tool *MonSTER* (*Mon*oid-based

*St*atic Analys*er*) developed to identify problematic patterns in Android apps which may pose security, privacy or functionality concerns. In this paper, we use this methodology to verify the apps studied in Section 3.1 adhere to the basic requirements needed to function as contact tracing tools, based on the responsibilities discussed in Section 2.1.1.

Although we apply this three-pronged methodology, consisting of manual, general static and bespoke static analysis, to contact tracing apps, we stress this methodology is highly customisable and can be applied to many domains. The manual analysis acts as a research stage, providing us with an understanding of the inner workings of a set of related apps. During the general static analysis stage, we screen the apps for common vulnerabilities that could occur in any domain. We also ensure the app is not malware. We then apply the knowledge gained during our manual analysis to our bespoke analysis stage, allowing us to search for design vulnerabilities that are unique to the domain in a repeatable, automated manner.

3.1 Collection of Apps

We chose a set of Android apps using the GAEN framework with open-source code, with the exception of *Protect Scotland* which is partially open-source. Table 1 shows a list of the apps, their country of origin or developer, the analysed version, the primary language used and the size of the code. These were downloaded on the 28th July 2020, except for *Stop-COVID-19* and *NHS Test & Trace*, which were downloaded on the 12th August 2020, and *Protect Scotland*, which was downloaded on the 10th September 2020. *SwissCovid* is developed using the DP3T protocol: this is extremely similar to the GAEN framework and uses the GAEN API as part of its design. We had to build some apps from source and disable ProGuard in order to generate unobfuscated APK files.

3.2 Manual Analysis

Having collected a series of apps, we began a manual analysis process. First, we ran the apps and systematically iterated over all possible functionality, with the exception of the later stages of the key submission process. Following this, we began a code review, plotting out the general structure of each app and noting how they interacted with the GAEN API. We achieved this by finding where the app calls the various GAEN API methods and following their respective call flows through the application. Finally, we reviewed any publicly released documentation.

3.3 General Static Analysis

We evaluated several Android static analysis tools as options for conducting off-the-shelf static analysis as might be used by a professional security analyst or penetration tester. Many tools are available freely and as commercial products; our point was to select something typical which demonstrates the capability of general static analysis tools in the context of our methodology, rather than find some "ultimate" best possible tool. We looked on GitHub and considered popularity as measured by GitHub stars. The most popular tools were MobSF (Abraham et al., 2016) and QARK (LinkedIn, 2015). Ultimately, we chose MobSF, since QARK flagged many trivial issues.

MobSF flags many generic Android security problems, such as certificate issues, hard-coded API keys and blacklisted malicious domains. It provides a useful condensed overview of the app including measurements such as the permissions used, the included native code libraries and the number of components — including exported components which extend an app's attack surface. Finally, MobSF summarises the overall code quality with an app security score, ranging from 0 to 100. We ran MobSF on all of our apps; the results are summarised in Section 5.2.

3.4 Bespoke Static Analysis - MonSTER

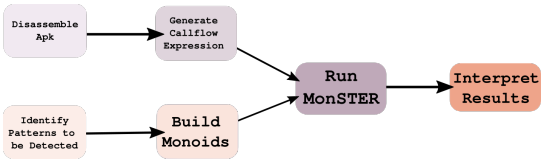

Figure 4: MonSTER workflow.

MonSTER is a static analysis tool written in Haskell and Python, using Androguard (Desnos et al., 2015). It can be configured to detect patterns of method calls in Android apps. These patterns are customisable to ensure certain liveness and safety properties are present in an app. It is intended to function as a tool to aid the testing of apps during and after development, using bespoke patterns to detect desirable or undesirable properties.

MonSTER uses a control-flow abstraction. A program is a collection of recursive procedures $f = e_f$ where f is a procedure identifier and e_f is an expression in the grammar:

$$e \quad ::= \quad a \mid f \mid e_1; e_2 \mid e_1 ? e_2.$$

Here a is an atomic procedure, ; and ? are sequential composition and non-deterministic branching.

Table 1: Contact tracing apps.

App Name	Origin	Version	Language	Code Size	URL
ApturiCovid	Latvia	1.0.47	Kotlin	313KB	apturicovid.lv
Corona-Warn-App	Germany	1.0	Kotlin	650KB	coronawarn.app/
Covid Safe Paths	MIT	None	TypeScript	2.4MB	pathcheck.org
CovidShield	Shopify	None	TypeScript	790KB	covidshield.app
Covid Tracker App	Ireland	1.0.4	TypeScript	430KB	covidtracker.gov.ie
Immuni	Italy	1.0.3	Kotlin	850KB	immuni.italia.it
NHS Test & Trace	UK	3.0	Kotlin	570KB	github.com/nhsx/
Protect Scotland	Scotland	1.0.0.30	TypeScript	Unknown	protect.scot
ProtegoSafe	Poland	1.0	Kotlin	500KB	gov.pl/web/protegosafe
Stop-Covid-19	Croatia	1.0	Java	230KB	github.com/stop-covid-19-croatia
Stopp Corona	Austria	1.2.0	Kotlin	860KB	github.com/austrianredcross
SwissCovid	Swiss	1.0.4	Java	520KB	https://github.com/DP-3T/

MonSTER converts method calls from Dalvik bytecode in an APK into this expression language. Branch points are abstracted by considering exit points of basic blocks as potential branches. Methods whose body can be ignored — such as API calls or those we aren't interested in — are treated as atomic methods and the rest are identifiers with definitions. This gives a set of expressions defining the overall program's execution, called the *call flow expression form* of the APK. Methods from this expression form are lifted into a monoid, picking out ones of interest.

Consider the first GAEN check as an illustration. For many of the analysed apps, before starting the GAEN client, there is a check to see it is already running. If it is, the function exits gracefully. The GAEN documentation is unclear as to what happens when the API is started if already running and whether this causes unexpected behaviour, such as the "resetting" of the protocol, inhibiting its effectiveness. So we treat the already-running check as good practice and will use MonSTER to verify that it takes place.

```
data StartCheck = U | ENABLED | START
                | ENABLED2START
                | START2ENABLED
                deriving (Eq, Ord, Show)
instance Mon StartCheck where
  unit = U
  mult c U = c
  mult U c = c
  mult ENABLED ENABLED = ENABLED
  mult START START = START
  mult START2ENABLED ENABLED = START2ENABLED
  mult START _ = START2ENABLED
  mult _ _ = ENABLED2START
  lift "ENABLED" = ENABLED
  lift "START" = START
  lift _ = U
```

Listing 1: Monoid for Start Check.

We wish to verify that a call to `isEnabled()` is followed by a call to `start()`. In Listing 1, the monoid's operator `mult` models method sequencing. To keep the monoid's policy clear and succinct, we use *keywords* to stand for groups of methods — for instance, every method considered to be a network sink is modelled by **SINK**. In this case, we replace all appearances of the `isEnabled()` and `start()` methods with the keywords **ENABLED** and **START**. When parsing the call flow expression, these keywords are embedded into the monoid using the `lift` operator shown in Listing 1. Methods that are not of interest are lifted to the monoid's identity. By creating such policies as monoids, we can define desirable or undesirable patterns customised to an app or app type.

Once translated into the monoid, MonSTER generates a system of equations using the program's call flow expression, consisting of a type expression for each procedure. MonSTER solves this system of equations by calculating its least fixed point. We expect to see the element **ENABLED2START** appear in the output of MonSTER only in cases where this pattern occurs.

MonSTER is kept simple by design: it focuses on an app's call flow and ignores its data flow. But, crucially, it can capture call flows for continually-executing code in its model of mutually recursive Büchi automata. So we can model the Android activity lifecycle, including implicit invocations (e.g., `onCreate()` followed by `onStart()`) as well as cycles (`onStop()` followed by `onResume()` then `onStart()` again). Liveness and safety properties can be ensured regardless of how the app is used, which is useful for checking longer term API call sequences such as used in GAEN. As Section 1 mentioned, secure APIs can be vulnerable to API fuzzing attacks: a string of API calls in a certain order leads to a security issue. Such vulnerabilities, once discovered, could be captured as custom rules in MonSTER.

4 MonSTER CHECKS

In this section, we outline the call flow patterns we aim to discover, alongside the example in Section 3.4. These patterns are intended to act as sanity checks, allowing a developer to verify an app meets the API requirements and is functional. Although we apply these checks to contact tracing apps, the methodology is highly customisable, allowing similar checks to be performed on a wide variety of apps. We base these checks around the necessary operations discussed in Section 2.1.1. To ensure accuracy, we also test a modified version of the Google reference app [2] designed to fail all of the checks.

4.1 Registration of Broadcast Receiver

The GAEN documentation states apps must register a Broadcast Receiver that handles the `ACTION_EXPOSURE_STATE_UPDATED` intent. This intent is broadcast when the user's exposure status has changed. The documentation contains a recommended way of doing this, as seen in Listing 2.

```
public void onReceive(Context c, Intent i) {
...
  if (ACTION_EXPOSURE_STATE_UPDATED.equals(action))
  {
    String token = i.getStringExtra(EXTRA_TOKEN);
    workManager.enqueue(
      new OneTimeWorkRequest.Builder(Update.class)
        .setInputData(new Data.Builder()
        .putString(EXTRA_TOKEN, token)
        .build()).build());
  }
}
```

Listing 2: Broadcast Receiver.

```
instance Mon BRCheck where
  ...
  mult GSE GSE = GSE
  mult OTWR OTWR = OTWR
  mult BUILD BUILD = BUILD
  mult ENQUEUE ENQUEUE = ENQUEUE
  mult _ GSE_OTWR = GSE_OTWR
  ...
  mult GSE_OTWR_BUILD ENQUEUE
       = GSE_OTWR_BUILD_ENQUEUE
  ...
```

Listing 3: Monoid for Receiver Check.

We implement a check to verify whether apps follow this recommendation. We identify the methods that compose this pattern: `getStringExtra()`, `OneTimeWorkRequest<init>()`, `build()` and `enqueue()` and replace them with the keywords **GSE**, **OTWR**, **BUILD** and **ENQUEUE**. As we are interested in a specific pattern, we do not need to define our multiplication rules fully, treating irrelevant situations as having no effect. Defining the monoid this way, we produce an element that represents the behaviour we are hoping to express: **GSE_OTWR_BUILD_ENQUEUE**. If this element is in MonSTER's output then the pattern is present in the app.

[2] https://github.com/google/exposure-notifications-android

4.2 Handling of Keys

We introduce another common sense check to ensure the apps are correctly managing their keys. This consists of two parts: the handling of the TEKs and the handling of the Diagnosis Keys. For the TEKs, we verify they are accessed only to be submitted to some central server i.e retrieving the keys is always followed by a network sink. Similarly, for the Diagnosis Keys, we ensure they are downloaded from a central server and then provided to the API i.e providing the keys is always preceded by a network source. To do this, we build a monoid consisting of the GAEN API methods `getTemporaryExposureKeyHistory()`, `provideDiagnosisKeys()` and all network sinks/-sources. We encode these as **RECENTKEYS**, **PROVIDEKEYS** and **NETWORK**. We also introduce an element **DOUBLE_SHARE** that allows us to see if there are multiple paths through the app that lead to TEK sharing.

```
instance Mon KeyCheck where
  ...
  mult RECENTKEYS RECENTKEYS = RECENTKEYS
  mult PROVIDEKEYS PROVIDEKEYS = PROVIDEKEYS
  mult NETWORK NETWORK = NETWORK
  mult NETWORK PROVIDEKEYS = SUBMIT
  mult RECENTKEYS NETWORK = SHARE
  mult SHARE SHARE = DOUBLE_SHARE
  ...
```

Listing 4: Monoid for Key Submission.

4.3 Notifying Users of Exposure

When a user becomes potentially infected after being exposed to an infected passerby, the app should inform the user via a push notification. Again, we can modify our tool in order to test whether this happens across all of the apps tested.

We use the monoid displayed in Listing 5 to test whether a notification is created after the `ACTION_EXPOSURE_STATE_UPDATED` broadcast receiver is triggered. We introduce a fictional method

RECEIVE to the start of the broadcast receiver's `onReceive()` method in the call flow expression form of each app. Furthermore, we replace any methods that create a push notification — such as `NotificationCompat.Builder()` — with the keyword **NOTIFY**. In particular, we are hoping we see the element **RECEIVE_NOTIFY** in the output.

```
instance Mon NotifyCheck where
  ...
  mult RECEIVE RECEIVE = RECEIVE
  mult NOTIFY NOTIFY = NOTIFY
  mult RECEIVE NOTIFY = RECEIVE_NOTIFY
  mult NOTIFY RECEIVE = NOTIFY_RECEIVE
  ...
```

Listing 5: Monoid for User Notification.

4.4 Updating the UI Correctly

Although not mentioned in the GAEN documentation, ensuring the user interface accurately reflects the state of the GAEN client is important. For instance, if an app stopped sharing TEKs but failed to indicate this, the user would reasonably assume they have a greater level of protection against Covid-19 than in reality. Similar problems may occur with privacy-conscious users unwittingly sharing TEKs.

The UI should update regardless of the entrypoint into the app. This problem can easily be represented in MonSTER by encoding each stage of the Android app lifecycle as a method. These methods can then call the methods of other stages in the lifecycle that are immediately reachable. This technique embeds all potential paths through the Android app lifecycle in the app's call flow expression, allowing us to check whether a property occurs in any possible path.

Following these preparations, we build our monoid as before. We are hoping to see a call to `isEnabled()` followed by a a call to any function that changes the UI, which we treat as a single class of methods. We replace these with **ENABLED** and **UI_CHANGE** respectively. The desired pattern is represented by the element **UPDATED**.

```
instance Mon UICheck where
  ...
  mult ENABLED ENABLED = ENABLED
  mult UI_CHANGE UI_CHANGE = UI_CHANGE
  mult ENABLED UI_CHANGE = UPDATED
  mult UI_CHANGE ENABLED = UI2ENABLED
  ...
```

Listing 6: Monoid for UI Update.

5 RESULTS

In this section, we discuss the results of the three stages of our analysis.

5.1 Manual Analysis

All of the apps tested could be described as wrappers around the GAEN API of various sizes and complexity, with the exception of *Covid-Safe-Paths* which has a GPS tracing mode. Almost all the activities of these apps are static and there are few ways the user can input arbitrary data; users can often only enter a random identification number to confirm they have been tested. There is no link between the user's identity and the identification number, which is provided in person at a medical centre.

5.1.1 Permissions & Services

In many of our apps, there is a failure to accurately convey the services and permissions needed for the GAEN client to operate. On Android, apps must request the Internet and Bluetooth permissions. They do not need any location permissions. However, Android requires Bluetooth, location and network *services* are active for the GAEN to function. In Google Play reviews, several users express confusion over this distinction, questioning why an app needs location services when it claims not to be tracking users.

During our manual analysis, we found several apps exacerbate this problem by failing to properly check whether the needed services are running when turning on exposure notifications, namely *Apturi Covid*, *Corona-Warn-App*, *Covid Tracker App* and *Protect Scotland*. All of these apps indicate exposure notifications are active when the GAEN client is blocked at an OS level due to some services not running. *Corona-Warn-App* and *Protect Scotland* trigger a notification asking the user to activate the required services but this is unreliable and can be dismissed. Otherwise, users are only informed of this problem in their phone's settings menu. This could easily impact the effectiveness of the GAEN protocol and user safety as user's may be misinformed to their level of protection against COVID-19.

5.1.2 Individual App Comments

Apturi Covid. If the user hands the responsibility of managing the GAEN client from this app to another app, *Apturi Covid* still indicates it is seeking nearby keys when, in actuality, it isn't. If the user then deactivates exposure notifications in the other app, *Apturi Covid* still indicates it is actively working, when no

exposure notifications are being sent. We think this is a significant safety problem, as a user could be misled into believing they were protected by the app when nothing was happening. We notified the developers of *Apturi Covid* of this and received a response stating they would fix this issue.

Corona-Warn-App. This is one of three apps whose functionality was questionable. When a user switches between contact tracing apps, this app throws a Java exception error. The error states the GAEN API is not active, although it actually is; it's just not being managed by *Corona-Warn-App*. Although minor, this error together with other problems discussed in this section may exacerbate confusion about the GAEN client's functionality.

Covid-Safe-Paths. This app has two distinct functionalities, GPS tracing and BLE tracing. Therefore, its attack surface is larger than other apps tested and it collected more sensitive information. Although users could delete their GPS location history from the app, there was no way from within the app to turn off this feature. An updated version of *Covid-Safe-Paths* has split its functionality into two apps, a GPS tracing app and a BLE tracing app.

Covid Tracker App. This app was donated to the Linux Foundation Public Health group; modified versions of it may be developed for other countries. The Linux Foundation will take care of the app's maintenance. The app allows the user to supply a phone number, but this number is stored on the phone and is only shared with the health authority if the user tests positive and provides permission. *Covid Tracker App* collects anonymised metric data, such as whether the user deleted the app during the onboarding process, but users must opt-in to this service.

NHS Test & Trace (England/Wales). Unlike the other apps here, this app allows users to scan QR codes outside of public spaces to provide the app with a rough user location. Our analysis was limited because a beta key was necessary to access the app's full functionality, which we did not get access to.

Stopp Corona. As with *Apturi Covid*, switching between apps causes problems with this app. As tracing apps act passively in the background, we feel this could be a significant problem as a user rarely interacting with *Stopp Corona* app may be inadvertently unprotected. We notified the developers of *Stopp Corona* of this but received no response.

SwissCovid. *SwissCovid* is produced by the researchers behind DP3T and can be seen as its reference implementation.We found the information and notifications shown to the user to be of a high standard compared to other apps, even warning users of the risk of linkage attacks when submitting their keys.

5.2 General Static Analysis

We outline the results of our MobSF scans in Table 2: the number of exported components, the number of potentially dangerous permissions requested, the certificate signing schemes used, the number of tracking libraries used and the code score — a score MobSF generates to surmise the code quality, ranging from 0 (worst) to 100 (best).

The OWASP Mobile Security Testing Guide (MSTG) [3] suggests these checks are relevant:

- exported components form an attack surface that may be exploited by malware. Thus, exported components could cause future vulnerabilities;
- tracking libraries introduce potential privacy violations of user data;
- insufficient signing schemes prevent developers from rotating their signing keys;
- additional permissions represent the capability for the phone to access user data or undertake "risky" actions.

For the number of exported components, *Covid Safe Paths* exceeds the other examples by some margin, demonstrating a concerning attack surface.

Due to the privacy-focus of these apps, most use no tracking libraries. Of the apps that use tracking libraries, these consist of Google Crashlytics, Firebase Analytics and, in the case of *Stop-Covid-19*, Google Admob. All of these libraries allow for the harvesting of information that is tangential to the operation of the GAEN API and could harm user privacy.

For APKs downloaded from the Play Store, the certificate signing schemes used are shown. MSTG recommends using all three schemes in apps that target modern Android SDK levels. Only 3 apps do this.

For apps using GAEN, the minimum required permissions are Bluetooth and Internet; some apps request much more than is necessary, particularly *Covid Safe Paths* and *Stopp Corona*. The version of *Covid Safe Paths* analysed includes GPS tracking functionality, which accounts for GL, but the other permissions seem unnecessary when contrasted with other apps. Similarly, *Stopp Corona* requests the usage of Bluetooth Admin, which is strongly discouraged by the GAEN documentation, and location services and is thus over-privileged. Permission creep is a well-established problem in Android (Vidas et al., 2011) and the principle of least privilege is considered good practise by MSTG. *Corona-Warn-App* and *NHS Test & Trace* require the use of the phone's camera (CA) as both of these apps utilise QR code scanning.

[3] https://github.com/OWASP/owasp-mstg/

Table 2: MobSF summary of Apps in Table 1.

App Name	Components	Additional Permissions	Certificate	Trackers	Score
Apturi Covid	4	WL	v1, v2	2	65
Corona Warn	3	WL CA	v1, v2, v3	0	50
Safe Paths	11	WL NL GL DS AA DS AR AA ST AT	None	1	5
CovidShield	4	WL AW	None	0	45
Covid Tracker	4	None	v1,v2	0	30
Immuni	3	WL	v1, v2	0	35
NHS Test & Trace	6	WL CA	v1,v2,v3	0	60
Protect Scotland	6	None	v1, v2, v3	0	30
ProtegoSafe	5	WL	v1, v2	2	70
Stop-Covid-19	3	WL	v1,v2,v3	3	90
Stopp Corona	3	WL BA NL LB	v1,v2	0	35
SwissCovid	3	WL	v1,v2	0	45

Key: WL: Wake Lock / BA: Bluetooth Admin / CA: Camera / AW: Alert Window
NL: Network Location / GL: GPS Location / LB: Location in Background / DS: Device Sync
AR: Activity Recognition / AA: Account Authentication / ST: Device Storage / AT: Access OS Task List

As most functionality of the GAEN framework is provided by the API, apps only need to be a wrapper around it, preferably as thin as possible. Any additional functionality increases the risk of security, privacy or functionality issues. This is reflected in the code score of each app which, as can be seen by cross-referencing Table 1 with Table 2, is inversely correlated with the code size; *Covid-Safe-Paths* has the worst score and is largest while *Stop-Covid-19* has the best score and is smallest.

5.3 Bespoke Static Analysis

The output of MonSTER is a list of tuples containing a method name and the monoid elements that can be reached from that method. We confirm the existence of a pattern in an app's source code by the existence of the monoid element representing that pattern in the output. We summarise our results in Table 3.

Check 1 - Starting Tracing in a Suitable Manner. All apps tested met this requirement except for *ProtegoSafe* and *Stop-Covid-19*, indicating neither app checks whether the Exposure Notification client is running before starting.

Check 2 - Registering a Broadcast Receiver. All apps tested met this requirement, except *Stop-Covid-19*. Of those that passed, all but one followed the implementation listed in the Google documentation exactly. The app that failed was *Stop-Covid-19* which registered a broadcast receiver but did not do so in the manner described in the documentation.

Check 3 - Handling of Temporary Keys. We could not perform this test on *Covid Tracker App*, *CovidShield* and *Protect Scotland* as parts of this process are coded in TypeScript. For all other apps, we found all calls to `getTemporaryExposureKeys()` are followed by a network sink and all calls to `provideDiagnosisKeys()` are preceded by a network source. We also ensure keys are sent to a single sink when submitted. We surmise that if a user is presented with the option to share their keys, all apps tested submit them to only one Diagnosis Server. Similarly, we conclude that after retrieving the Diagnosis Keys from the server, these apps correctly provide them to the API.

Check 4 - Notifying Users of Exposure. Again, we could not run this check on *CovidShield* as it is largely written in Typescript. Furthermore, the heavy use of dependency injection in *Immuni* and *ProtegoSafe* limits MonSTER's ability to generate meaningful call flow expressions which hinders its ability to analyse these apps. Of the apps properly tested, only *Covid-Safe-Paths* failed.

Check 5 - Updating the UI Correctly. *Stopp Corona* and *Apturi Covid* failed this check. The logic for updating the UI in these apps is handled in the `onCreate()` method of the main landing page instead of `onResume()`. Thus, one can activate exposure notifications, close the app and turn them off — either in the phone's settings or using another contact tracing app — and neither app will update the UI, instead incorrectly informing the user that the app is contact tracing. Manual testing confirmed this behaviour.

As seen from Table 2 and Section 5.2, the results generated by MobSF, although worthwhile from a security perspective, reveal nothing about the usage of the GAEN framework. In contrast, our bespoke analysis with MonSTER allows us to generate strong guarantees about the functionality of the apps and their adherence to the GAEN requirements.

Although MonSTER requires more effort to produce customised checks, its advantages over general

Table 3: Results of our MonSTER checks.

App Name	1	2	3	4	5
Apturi Covid	✓	✓	✓	✓	X
Covid Tracker App	✓	✓	-	✓	✓
Corona-Warn-App	✓	✓	✓	✓	✓
CovidShield	✓	✓	-	-	-
Covid Safe Paths	✓	✓	✓	X	-
Immuni	✓	✓	✓	-	✓
Protect Scotland	✓	✓	-	✓	✓
ProtegoSafe	X	✓	✓	-	✓
NHS Test & Trace	✓	✓	✓	✓	✓
Stop-Covid-19	X	X	✓	✓	✓
Stopp Corona	✓	✓	✓	✓	X
SwissCovid	✓	✓	✓	✓	✓
Misconfigured App	X	X	X	X	X

static analysis are clear. MonSTER's call flow checking allows the user to fine-tune the properties to be checked that are unique to a given app or set of apps. Such properties can then be checked repeatedly throughout the app's development and release.

6 DISCUSSION

6.1 Limitations of Methodology

MonSTER is a prototype designed to explore our methodology and thus has issues. One pitfall is scalability; larger apps take far longer to analyse. We can mitigate this problem by excluding irrelevant code from the analysis. Moreover, we can only analyse patterns that appear in the app's bytecode i.e those written using Java or Kotlin. Finally, some programming constructs that rely on generated methods, such as Dependency Injection libraries and coroutine support, limit MonSTER's ability to build accurate call flow expressions, requiring manual fixing.

6.2 Discussion of Contact Tracing Apps

When evaluating apps for this paper, we found few we could properly analyse. Many tracing apps using the GAEN API are not open source and these all used code obfuscation. We believe this is counterproductive to the goals of the GAEN as the end-user has little guarantee of the app's capabilities and whether it has faithfully implemented the protocol. To instill greater trust in end-users that the apps are working as intended, providing a public verification method, such as open-source code or third-party audits, would be advantageous. In these times, accurately functioning contact tracing should take precedence over intellectual property.

7 RELATED WORK

Unlike our work which focuses on correct implementation, most current research on contact tracing apps focuses on the design of the underlying frameworks, particularly with respect to privacy. Cho et al. define three notions of privacy for contact tracing apps: privacy from snoopers, contacts, and the authorities (Cho et al., 2020). They note some information will always be revealed and simple attacks can always be performed; therefore an acceptable level of privacy should be defined with respect to these three parties. Gvili analyses privacy issues with the GAEN framework and proposes attacks that would hinder its effectiveness, such as relay and replay attacks (Gvili, 2020). Similarly, Magklaras et al. assess the weaknesses of published tracing frameworks(Magklaras and Bojorquez, 2020). Some research does focus on app implementations, but at a higher level compared to us. Samhi et el. provide a categorisation of existing apps on Google Play related to Covid-19, but do not analyse apps individually (Samhi et al., 2020).

There are many static analysing tools for the Android platform. Li et al. (Li et al., 2017) identified over 100 such tools. Unlike MonSTER, the majority of these tools establish an app is secure using a generalised ruleset. For instance, the MobSF (Abraham et al., 2016) and QARK (LinkedIn, 2015) work by analysing decompiled code and flagging bad programming practises that may lead to security issues, such as the existence of logging or API keys. Some tools are more specific but lacking the customisation of MonSTER; for instance, taint analysis research has led to tools such as FlowDroid (Arzt et al., 2014), designed to ensure sensitive information cannot be exfiltrated from an app. MonSTER can also be seen as a static analogue of dynamic analysis call tracing, utilised by tools such as DroitMat (Wu et al., 2012) and DroidTrace (Zheng et al., 2014). However, both of these tools focus on identifying malware, rather than functionality properties like MonSTER.

8 CONCLUSION

This paper presented an analysis into the functionality, security and privacy of contact tracing apps using a methodology involving manual, general static and bespoke static analysis. For the bespoke case, we present MonSTER, a lightweight, static analysis tool that can detect the existence of patterns of Android app behaviour in a customisable way, as general static analysis tools were not sufficient. Using this process, we verified that many contact tracing

apps adhered to the GAEN API's recommended usage. However, we found failings in tested versions of some apps that could impact user safety or security, namely *Covid-Safe-Paths*, which failed to adhere to design practices that minimise user risk, *Apturi Covid* and *Stopp Corona*, which failed to correctly inform users of the status of the GAEN client. For future work, we mention that MonSTER's generation of call flow expressions from an app's bytecode could be improved to capture more programming constructs, such as coroutines.

ACKNOWLEDGEMENTS

We are grateful for support for this work from the Office of Naval Research ONR NICOP award N62909-17-1-2065 and The Alan Turing Institute under the EPSRC grant EP/N510129/1.

REFERENCES

Abraham, A., Schlecht, D., Dobrushin, M., and Nadal, V. (2016). Mobile Security Framework (MobSF). https://github.com/MobSF.

Amnesty (2020). Bahrain, Kuwait and Norway Contact Tracing Apps among Most Dangerous for Privacy. https://www.amnesty.org/en/latest/news/2020/06/bahrain-kuwait-norway-contact-tracing-apps-danger-for-privacy/. Accessed: 2020-08-04.

Anomali (2020). Anomali Threat Research Identifies Fake COVID-19 Contact Tracing Apps Used to Download Malware that Monitors Devices, Steals Personal Data. https://www.anomali.com/blog. Accessed: 2020-09-10.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *Acm Sigplan Notices*, 49(6):259–269.

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., and Penix, J. (2008). Using Static Analysis to Find Bugs. *IEEE software*, 25(5):22–29.

Bortolozzo, M., Centenaro, M., Focardi, R., and Steel, G. (2010). Attacking and Fixing PKCS#11 Security Tokens. pages 260–269.

Cho, H., Ippolito, D., and Yu, Y. W. (2020). Contact Tracing Mobile Apps for COVID-19: Privacy Considerations and Related Trade-offs.

Desnos, A. et al. (2015). Androguard. https://github.com/androguard/androguard.

ESET (2020). New Ransomware Posing as COVID-19 Tracing App Targets Canada. https://www.welivesecurity.com/2020/06/24/. Accessed: 2020-09-10.

Google (2020). Exposure Notifications: Android API Documentation. https://web.archive.org/web/20200603200341/https://static.googleusercontent.com/media/www.google.com/en//covid19/exposurenotifications/pdfs/Android-Exposure-Notification-API-documentation-v1.3.2.pdf. Accessed: 2020-08-04.

Gvili, Y. (2020). Security analysis of the covid-19 contact tracing specifications by apple inc. and google inc.

Leith, D. J. and Farrell, S. (2020). Coronavirus Contact Tracing: Evaluating the Potential of using Bluetooth Received Signal Strength For Proximity Detection.

Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. (2017). Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology*, 88:67–95.

LinkedIn (2015). Quick Android Review Kit (QARK). https://github.com/linkedin/qark.

Magklaras, G. and Bojorquez, L. N. L. (2020). A Review of Information Security Aspects of the Emerging COVID-19 Contact Tracing Mobile Phone Applications.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. (2018). Lessons from Building Static Analysis Tools at Google. *Communications of the ACM*, 61(4):58–66.

Samhi, J., Allix, K., Bissyandé, T. F., and Klein, J. (2020). A First Look at Android Applications in Google Play related to Covid-19.

TCN (2020). TCN coalition. https://www.covid19.nhs.uk/.

Troncoso, C., Payer, M., Hubaux, J.-P., Salathé, M., Larus, J., Bugnion, E., Lueks, W., Stadler, T., Pyrgelis, A., Antonioli, D., et al. (2020). Decentralized Privacy-Preserving Proximity Tracing. *arXiv preprint arXiv:2005.12273*.

Vidas, T., Christin, N., and Cranor, L. (2011). Curbing Android Permission Creep. In *Proceedings of the Web*, volume 2, pages 91–96.

Wan, Z. and Liu, X. (2020). ContactChaser: A Simple yet Effective Contact Tracing Scheme with Strong Privacy. Cryptology ePrint Archive, Report 2020/630. https://eprint.iacr.org/2020/630.

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. (2012). Droidmat: Android Malware Detection through Manifest and API Calls Tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE.

Zheng, M., Sun, M., and Lui, J. C. (2014). DroidTrace: A Ptrace Based Android Dynamic Analysis System with Forward Execution Capability. In *2014 international wireless communications and mobile computing conference (IWCMC)*, pages 128–133. IEEE.